KMP算法

什么是KMP算法：

KMP是三位大牛：D.E.Knuth、J.H.Morris和V.R.Pratt同时发现的。其中第一位就是《计算机程序设计艺术》的作者！！

KMP算法要解决的问题就是在字符串（也叫主串）中的模式（pattern）定位问题。说简单点就是我们平时常说的关键字搜索。模式串就是关键字（接下来称它为P），如果它在一个主串（接下来称为T）中出现，就返回它的具体位置，否则返回-1（常用手段）。

首先，对于这个问题有一个很单纯的想法：从左到右一个个匹配，如果这个过程中有某个字符不匹配，就跳回去，将模式串向右移动一位。这有什么难的？

我们可以这样初始化：

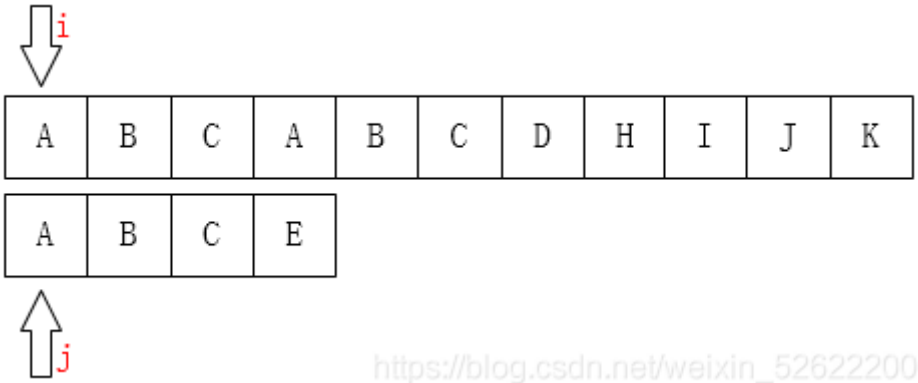

图1

之后我们只需要比较i指针指向的字符和j指针指向的字符是否一致。如果一致就都向后移动，如果不一致，如下图：

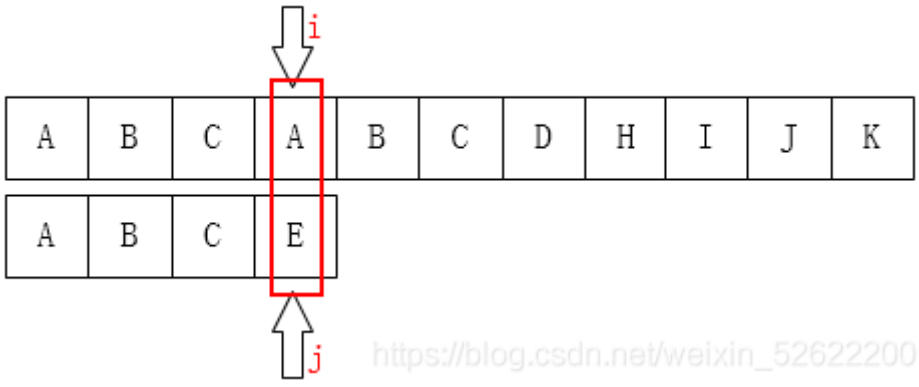

图2

A和E不相等，那就把i指针移回第1位（假设下标从0开始），j移动到模式串的第0位，然后又重新开始这个步骤：

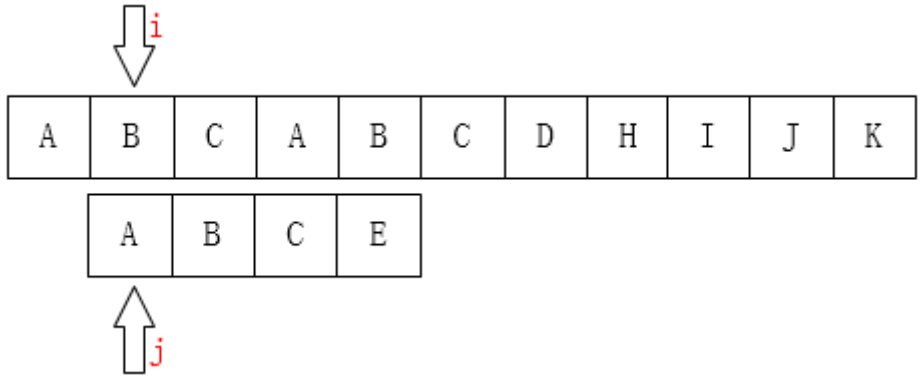

图3

```
char t[N],p[N];
scanf("%s",t);
scanf("%s",p);
int lent = strlen(t);
int lenp = strlen(p);
int i = 0; // 主串的位置
int j = 0; // 模式串的位置
while (i < lent && j < lenp) {
   if (t[i] == p[j]) { // 当两个字符相同，就比较下一个
       i++;
       j++;
   } else {
       i = i - j + 1; // 一旦不匹配，i后退
       j = 0; // j归0
   }
}
if (j == lenp) {
   return i - j;
} else {
   return -1;
}
```

上面的程序是没有问题的，但不够好！

如果是人为来寻找的话，肯定不会再把i移动回第1位，因为主串匹配失败的位置前面除了第一个A之外再也没有A了，我们为什么能知道主串前面只有一个A？因为我们已经知道前面三个字符都是匹配的！（这很重要）。移动过去肯定也是不匹配的！有一个想法，i可以不动，我们只需要移动j即可，如下图：

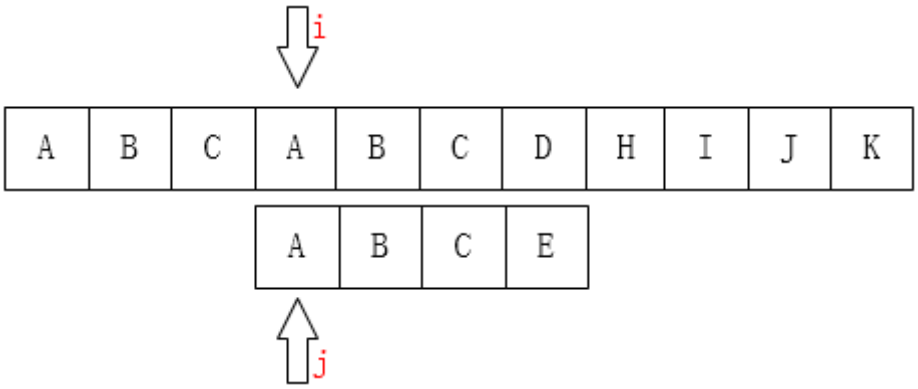

图4

上面的这种情况还是比较理想的情况，我们最多也就多比较了再次。但假如是在主串“SSSSSSSSSSSSSA”中查找“SSSSB”，比较到最后一个才知道不匹配，然后i回溯，这个的效率是显然是最低的。

大牛们是无法忍受“暴力破解”这种低效的手段的，于是他们三个研究出了KMP算法。其思想就如同我们上边所看到的一样：“利用已经部分匹配这个有效信息，保持i指针不回溯，通过修改j指针，让模式串尽量地移动到有效的位置。”

所以，整个KMP的重点就在于当某一个字符与主串不匹配时，我们应该知道j指针要移动到哪？

接下来我们自己来发现j的移动规律：

如图：C和D不匹配了，我们要把j移动到哪？显然是第1位。为什么？因为前面有一个A相同啊：

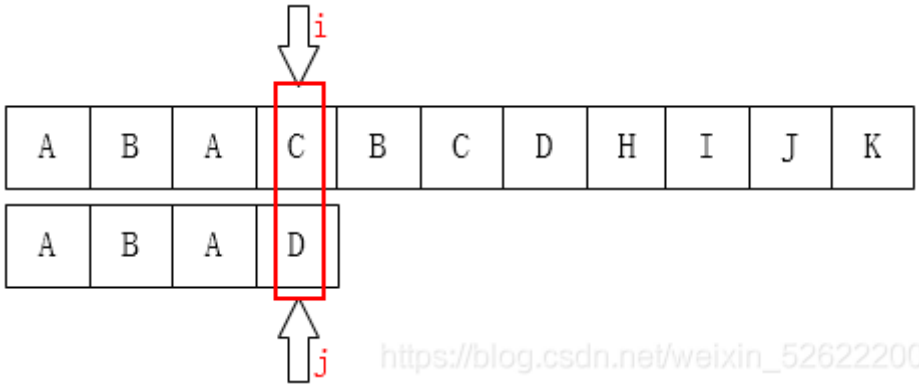

图5

如下图也是一样的情况：

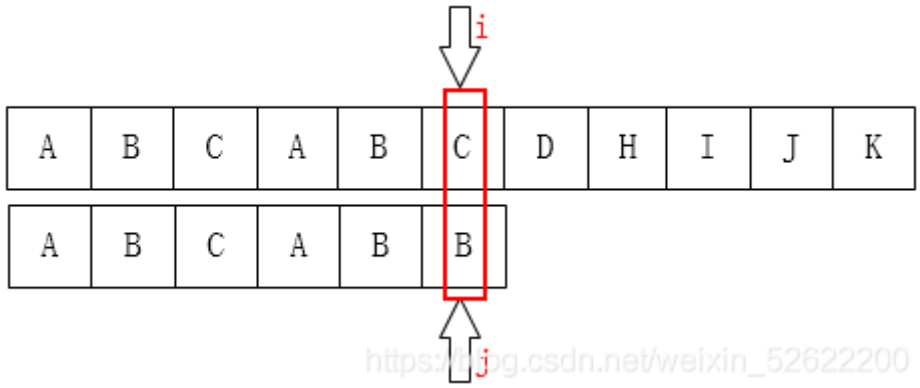

图6

可以把j指针移动到第2位，因为前面有两个字母是一样的：

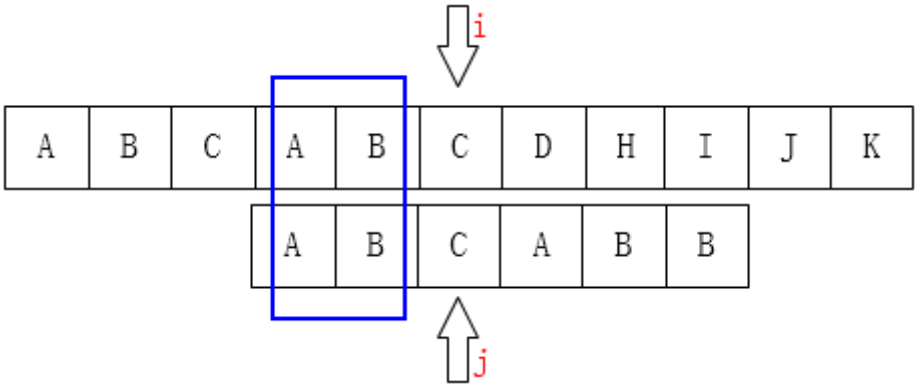

图7

如果用式子表示的话就是

```
P[0 ~ k-1] == P[j-k ~ j-1]
```

这个相当重要，如果觉得不好记的话，可以通过下图来理解：

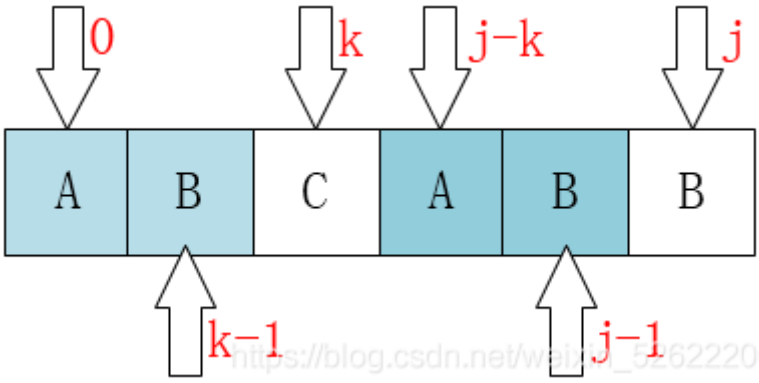

图8

弄明白了这个就应该可能明白为什么可以直接将j移动到k位置了。

```
当T[i] != P[j]时 //匹配的下一个位置不一样
有T[i-j ~ i-1] == P[j-k ~ j-1]  //主串的前一些位置和模式串是相同的
由P[0 ~ k-1] == P[j-k ~ j-1]
必然：T[i-k ~ i-1] == P[0 ~ k-1]
```

好，接下来就是重点了，怎么求这个（这些）k呢？因为在P的每一个位置都可能发生不匹配，也就是说我们要计算每一个位置j对应的k，所以用一个数组next来保存，next[j] = k，表示当T[i] != P[j]时，j指针的下一个位置。

```
void get_next(){
    int i = 0;
    int j;
    j = next[0] = -1;
    while(i < p){
        while(j != -1 && p[j] != p[i]){
            j = next[j];
        }
        next[++i] = ++j;
    }
}
```

这个版本的求next数组的算法应该是流传最广泛的，代码是很简洁。可是真的很让人摸不到头脑，它这样计算的依据到底是什么？

好，先把这个放一边，我们自己来推导思路，现在要始终记住一点，next[j]的值（也就是k）表示，当P[j] != T[i]时，j指针的下一步移动位置。

先来看第一个：当j为0时，如果这时候不匹配，怎么办？

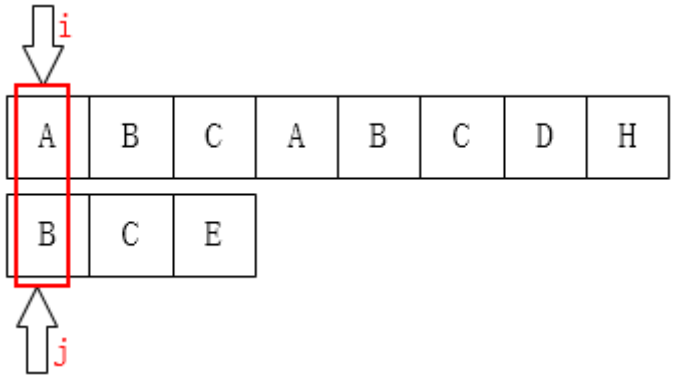

图9

像上图这种情况，j已经在最左边了，不可能再移动了，这时候要应该是i指针后移。所以在代码中才会有next[0] = -1;这个初始化。

如果是当j为1的时候呢?

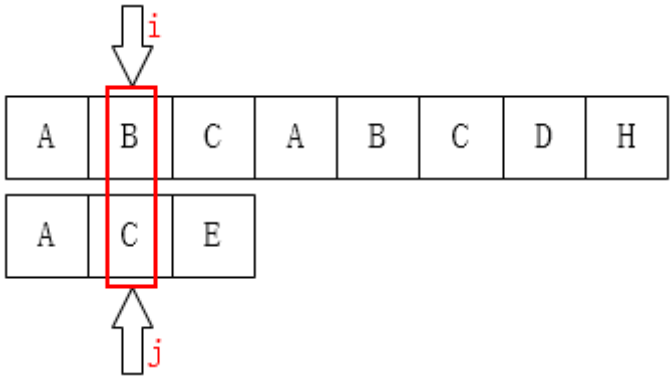

图10

显然，j指针一定是后移到0位置的。因为它前面也就只有这一个位置了~

下面这个是最重要的，请看如下图：

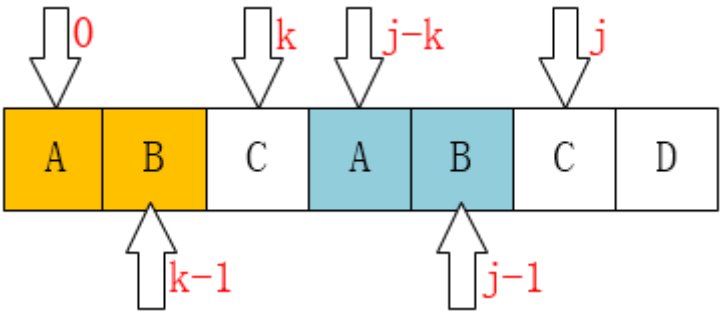

图11

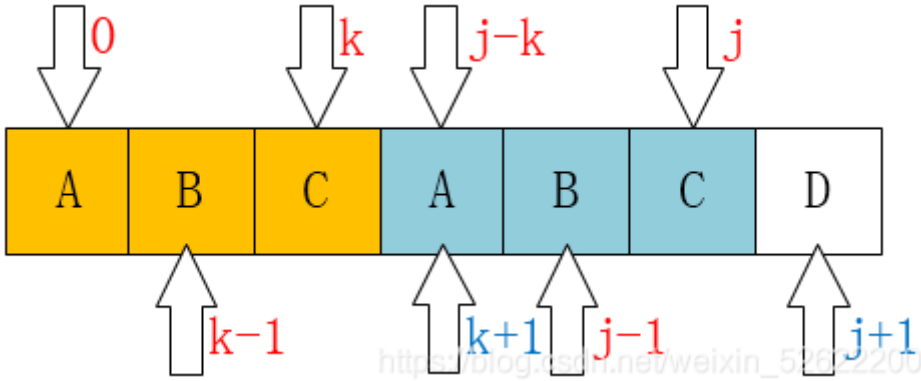

图12

请仔细对比这两个图。

我们发现一个规律：

```
当P[k] == P[j]时，
有next[j+1] == next[j] + 1
```

证明：

```
因为在P[j]之前已经有P[0 ~ k-1] == p[j-k ~ j-1]。（next[j] == k）
这时候现有P[k] == P[j]，我们是不是可以得到P[0 ~ k-1] + P[k] == p[j-k ~ j-1] + P[j]。
即：P[0 ~ k] == P[j-k ~ j]，即next[j+1] == k + 1 == next[j] + 1。
```

这里的公式不是很好懂，还是看图会容易理解些。

那如果P[k] != P[j]呢？比如下图所示：

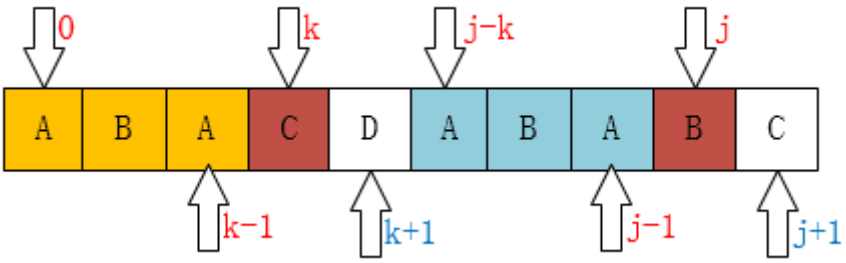

图13

像这种情况，如果你从代码上看应该是这一句：k = next[k];为什么是这样子？你看下面应该就明白了。

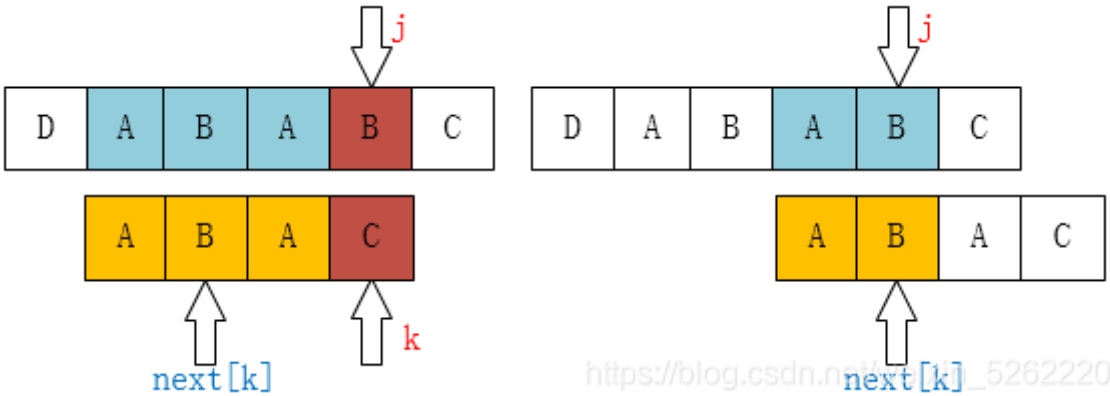

图14

现在你应该知道为什么要k = next[k]了吧！像上边的例子，我们已经不可能找到[A，B，A，B]这个最长的后缀串了，但我们还是可能找到[A，B]、[B]这样的前缀串的。所以这个过程像不像在定位[A，B，A，C]这个串，当C和主串不一样了（也就是k位置不一样了），那当然是把指针移动到next[k]啦。

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;
char t[1000005],p[1000005];
int next2[1000005];
int lt,lp;
void get_next(){
    int i = 0;
    int j;
    j = next2[0] = -1;
    while(i < lp){
        while(j != -1 && p[j] != p[i]){
            j = next2[j];
        }
        next2[++i] = ++j;
    }
}
int kmp(){
    int i = 0,j = 0,sum = 0;
    get_next();
    while(i < lt){
        while(t[i] != p[j] && j != -1){
            j = next2[j];
        }
        ++i;
        ++j;
        if(j == lp){
            printf("%d\n",i - j + 1);
            j = next2[j];
        }
    }
    return 0;
}
int main(){
    scanf("%s",t);
    scanf("%s",p);
    lt = strlen(t);
    lp = strlen(p);
    kmp();

    for(int i = 1; i <= lp; i++){
        if(i != 1) printf(" ");
        printf("%d",next2[i]);
    }
    return 0;
}
```